Technical Report No. 99-433

Nonlinearity, Maximization, and Parallel Real-Time Computation*

Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada
Email: akl@cs.queensu.ca

May 10, 2000

Abstract

This paper focuses on the improvement in the quality of computation provided by parallelism. The problem of interest is that of computing the maximum of a nonlinear feedback function in a real-time environment. We show that the solution obtained in parallel is significantly, provably, and consistently better than a sequential one. It is important to note that our purpose is not to demonstrate merely that a parallel computer can obtain a better solution to a computational problem than one derived sequentially. The latter is an interesting (and often surprising) observation in its own right, but we wish to go further. It is shown here that the improvement in quality can be *arbitrarily high*. To be specific, the ratio of the parallel solution to the sequential one is typically *superlinear* in the number of processors used by the parallel computer. This result is akin to superlinear speedup—a phenomenon itself originally thought to be impossible.

Key words and phrases: Parallelism, real-time computation, nonlinear feedback function, maximization.

*This research was supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The central motivation behind parallelism has always been the speeding up of sequential computations. Recently, another aspect of parallel computation was brought to light. It was shown that under some circumstances it is possible to obtain *in parallel* solutions to computational problems that are significantly better than any solutions computed sequentially. This phenomenon was demonstrated for problems in combinatorial optimization, cryptography, numerical computation, and statistical analysis [4, 5, 6].

This paper provides a different example of a computation in which the results arrived at in parallel offer an asymptotic improvement over their sequential counterparts. The computation we describe falls within the *real-time paradigm.* Here, the data needed to solve a problem are received *on-line* and the results of the computation are to be delivered by a certain *deadline*. Specifically, we consider the problem of computing the maximum value of a *nonlinear feedback function* over a given interval in a real-time environment. It is shown that the solution reached in parallel is significantly better than the best sequential one. Furthermore, the improvement is provable and consistent.

It is worth stressing here that the purpose of this paper is not to demonstrate merely that a parallel computer can obtain a better solution to a computational problem than one derived sequentially. The latter is an interesting (and often surprising) observation in its own right, but we wish to go further. It is demonstrated in what follows that the improvement in quality can be *arbitrarily high.* To be specific, the ratio of the parallel solution to the sequential one is typically *superlinear* in the number of processors used by the parallel computer. This result is akin to *superlinear speedup*—a phenomenon itself originally thought to be impossible [1, 2, 3, 7, 12, 13, 14, 15, 16, 41, 42, 43].

The remainder of this paper is organized as follows. Some background material relative to *real-time computation* and *models of computation* is presented in Section 2. The application illustrating the ability of parallel computers to obtain solutions of higher quality than possible sequentially is the subject of Section 3. Concluding remarks are offered in Section 4.

2 Background

In this section we introduce the real-time paradigm as well as the models of computation used in this paper. Henceforth, we adopt the standard definition of *time unit*, that is, the unit traditionally used to measure the running time of an algorithm [1, 11, 23, 34]: A time unit is the length of time required by a processor to read a datum from memory, perform a constant-time operation (such as adding two numbers), and write a datum to memory.

2.1 Real-Time Computation

The prevalent paradigm of computation, to which everyone who uses computers is accustomed, is one in which all the data required by an algorithm are available when the computer starts working on the problem to be solved. A different paradigm is *real-time* computation. Here, not all inputs are given at the outset. Rather, the algorithm receives its data (one or several at a time) *during* the computation, and must incorporate the newly arrived inputs in the solution obtained so far. Often, the data-arrival rate is constant; specifically, $\mathcal{N}$ data are received every $\mathcal{T}$ time units, where both $\mathcal{N}$ and $\mathcal{T}$ are fixed in advance.

A fundamental property of real-time computation is that certain operations must be performed by specified deadlines. Thus, one or more of the following conditions may be imposed:

1. Each received input (or set of inputs) must be processed within a certain time after its arrival.

2. Each output (or set of outputs) must be returned within a certain time after the arrival of the corresponding input (or set of inputs).

Thus, for example, it may be crucial for an application that each input be operated on as soon as it is received. Similarly, each partial solution (as well as the final one) may need to be returned as soon as it is available [29, 36, 48]. It is helpful to note here that, when no deadlines are imposed, computations for which inputs arrive while the algorithm is in progress are referred to as *on-line* [25, 30, 32, 33], *incremental* [19, 20, 44, 49], *dynamic* [9, 10, 53], and *updating* [18, 21, 26, 35, 46, 47, 50, 52]. It is also important to note that our definition, while striving to be as general as possible, is particularly suitable for our purposes in this paper. Many other definitions exist; see, for example, the various interpretations of the notion of real time provided in [8, 38, 51].

2.2 Models of Computation

Two models, one sequential and one parallel, are applied to the solution of the various real-time computational problems studied in this paper. At the outset we state explicitly the following basic assumption: The analyses in this paper assume that all models of computation are the fastest possible (within the bounds established by theoretical physics). Specifically, no machine exists that is faster than the sequential computer of Section 2.2.1, and similarly no parallel computer exists whose processors are faster than the processors of the parallel computer of Section 2.2.2. This is the fundamental assumption in parallel computation. One should also keep in mind here that the length of a time unit is not an absolute quantity. Instead, the duration of a time unit is defined in terms of the speed of the processors available (namely, the single processor on the sequential computer and each processor on the parallel machine).

2.2.1 Sequential Model

This is the conventional model of computation used in the design and analysis of sequential (or *serial*) algorithms. It consists of a single processor equipped with a random-access memory to which the processor can gain access for the purpose of reading and writing. The processor has some local registers for intermediate results, a control memory to store its program, and some circuitry to perform arithmetic and logical operations. It also has input and output devices for communication with the outside world. During each cycle of the computation, the processor executes one instruction from its program: It fetches a datum from memory, performs an operation on it, and stores the result back in memory.

2.2.2 Parallel Model

Our chosen parallel model is the *pipeline* computer, shown in Fig. 1 [1]. In this model, n processors, denoted by P_1, P_2, $\ldots$, P_n, are connected to one another by (one-way) communication links such that:

1. P_1 receives its input from (and only from) the outside world.

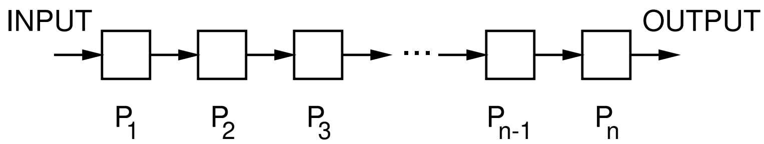

Figure 1: Parallel computer.

2. P_i receives its input from (and only from) P_{i-1}, $2 \leq i \leq n$.

3. P_i sends its output to (and only to) P_{i+1}, $1 \leq i \leq n-1$.

4. P_n sends its output to (and only to) a memory or a communications channel.

Data travel from P_1 to P_n, with P_i beginning to operate only when it receives input, $1 \leq i \leq n$. Each processor is of the type described in Section 2.2.1. It can be argued that the pipeline computer is the weakest of all models of parallel computation in which the processors have some means of communicating among themselves. Nonetheless this model, with its rudimentary communication paths, is perfectly suitable when solving the real-time computational problems of this paper. This is demonstrated in Section 3, where it is shown that the pipeline computer affords a parallel algorithm that is significantly better than a sequential one.

3 Maximizing Nonlinear Feedback Functions

Let f be a function of some real (or complex) variables. It is frequently necessary to find an optimal value of f, subject to a number of conditions. Here, f is called the *objective function* and the conditions are known as the *constraints*. The optimal value is typically a *maximum* or a *minimum* of f satisfying the constraints. Often, when the exact maximum (or minimum) is difficult to obtain, an *approximation* of the optimal value is computed [37, 45].

Evidently, we are interested here in finding the optimal value of f in a real-time setting. Our purpose is to demonstrate the ability of a parallel algorithm to do better than the best sequential algorithm when computing the maximum of a nonlinear feedback function in real time. In this context, a *better* solution is one that is *closer to maximum*.

The computational paradigm used in this section is presented in Section 3.1 along with a definition of the specific problem to be solved. Sequential and parallel solutions and their analyses are developed in Sections 3.2, 3.3, and 3.4, respectively.

3.1 Real-Time Maximization

We begin by describing the specific computation chosen to illustrate our point. For ease of exposition, the objective function to be maximized, as well as the constraints, are kept as simple as possible. Thus, the computational problem to be solved calls for finding the maximum of a function of a single real variable, in a given range.

The function to be maximized is first presented. The real-time computational environment and the conditions under which the solution is to be obtained are then introduced.

3.1.1 Nonlinear Feedback Functions

The functions of interest in this paper and whose maximum is to be found in a given range are called *nonlinear feedback functions*. In some contexts, they are known as *aperiodic*, *chaotic*, and *complex functions* [17, 22, 24, 27, 31, 39, 40, 54]. These functions are typically defined recursively as follows.

Let x_0 be a real and n a positive integer. Thus, a sequence of real numbers $x_1, x_2, \ldots, x_n$ is obtained from the relation:

$$x_{i+1} = f({x_i}^b, i, n), \qquad \text{for} \;\; b > 1 \;\; \text{and all} \;\; i \geq 0. \tag{1}$$

Here, f combines ${x_i}^b$ and the constant n with various multiplicative and additive terms, as well as other simple arithmetic functions. Given f, x_0, and n, it is required to find the largest of $x_1, x_2, \ldots, x_n$.

Example 3.1 One example of such a function, which will prove particularly useful to our subsequent analysis, is:

$$x_{i+1} = \left[\Big(\lfloor x_i \rfloor + (-1)^{i+1}(i+1) \Big)^{2u} \bmod \left(n + (-1)^{i+1}\right)^v \right]^{w/v} \tag{2}$$

for $i \geq 0$, and positive integers u, v, and w, with $w > 1$.

Suppose for illustration that $u = 1$, $v = 2$ and $w = 3$. We have:

$$x_{i+1} = \left[\left(\lfloor x_i \rfloor + (-1)^{i+1}(i+1) \right)^2 \bmod (n + (-1)^{i+1})^2 \right]^{3/2}$$

for $i \geq 0$.

Taking, for instance, $x_0 = 14.0$ and $n = 10$, we get: $x_1 = 18.520259$, $x_2 = 225.06221$, $x_3 = 216.0$, $x_4 = 0.0$, $x_5 = 125.0$, $x_6 = 1000.0$, $x_7 = 216.0$, $x_8 = 742.54158$, $x_9 = 64.0$, $x_{10} = 172.60069$.

In other words, the x_i values oscillate unpredictably, and that particular x_i achieving the maximum cannot be guessed in advance. The only way to find the largest x_i is to compute $x_1, x_2, \ldots, x_n$. $\square$

For the purposes of this paper we make the following assumptions:

1. The objective function is of the form given in Equation (1).

2. The function f to be maximized consists of a constant number of terms (i.e., f can be expressed using no more than a certain number of symbols fixed in advance). Similarly, each of x_0 and n fits in a constant number of words in memory. It is to be noted, as a consequence, that the optimization problem to be solved, being defined by f, x_0, and n, has a constant size formulation.

3. Each of $x_1, x_2, \ldots, x_n$ (and, consequently, the maximum value of f) also fits in a constant number of words. This assumption and the previous one together imply that the size of f, n, x_i, $i \geq 0$, and the current maximum is a constant multiple of the word size in bits. Therefore, this quadruple can be transmitted and received in a constant number of time units.

3.1.2 Computing the Maximum in Real Time

The specific problem to be solved in this section is defined as follows:

1. A computer system receives a stream of input in real time. These inputs represent the data of an optimization problem.

2. Time is divided into *intervals*. Each interval is $\mathcal{T}$ time units long, where $\mathcal{T}$ is a constant.

3. At the beginning of the jth time interval, $j > 0$, an objective function f^j is received, together with a pair of constraints $C^j = (x_0^j, n^j)$.

4. It is required that the pair (f^j, C^j) be processed as soon as it is received and that the maximum value of $x_1^j, x_2^j, \ldots, x_n^j$ (or an approximation of it) subject to C^j be produced as output as soon as it is computed. Furthermore, one output must be produced at the end of each time interval (with possibly an initial delay before the first output is produced).

5. **Computational Assumption.** In one time interval a processor can

   (a) Read f^j, n^j, x_i^j, and the current maximum,

   (b) Compute x_{i+1}^j and the new maximum, and

   (c) Output f^j, n^j, x_{i+1}^j, and the new maximum.

We now provide sequential and parallel solutions to this problem. This is followed by a comparative analysis.

3.2 Sequential Solution

A function f^j and a pair of constraints C^j are received at the beginning of the jth time interval. These must be processed and the required maximum (or an approximation thereof) must be produced before the new function f^{j+1} and the new pair of constraints C^{j+1} are received (and demand to be processed) at the beginning of the $(j+1)$st time interval. A sequential computer, by definition, has only one processor. Conforming to the computational assumption, in one time interval, the processor

1. Receives f^j, x_0^j, and n^j,

2. Computes x_1^j using x_0^j, n^j, and the definition of f^j, and

3. Returns x_1^j as the required maximum.

Note here that x_1^j is not guaranteed to be the maximum of $x_1^j, x_2^j, \ldots, x_n^j$, as specified by the problem definition. Since the sequential computer cannot compute $x_2^j, x_3^j, \ldots, x_n^j$ before the pair (f^{j+1}, C^{j+1}) is received, it returns the *only* approximation of the maximum that it can obtain.

3.3 Parallel Solution

On a pipeline computer with n processors, P_1, P_2, …, P_n, processor P_1 is in charge of reading new inputs, while P_n is designated to produce the output. Therefore, at the beginning of the jth interval, P_1 receives (f^j, C^j). It computes x_1^j and (for lack of another value with which to compare it) calls it the current maximum. It then sends the quadruple $(f^j, x_1^j, n^j,$ current maximum) to P_2. The jth time interval has now ended and the $(j+1)$st commences. While P_1 is reading a new input, P_2 receives the quadruple sent by P_1. It computes x_2^j, compares it with current maximum, updates the latter if necessary, and sends the new quadruple $(f^j, x_2^j, n^j,$ current maximum) to P_3. This continues, with processor P_k computing x_k^j during time interval $j+k-1$, $j>0$, $k \geq 1$. The maximum of $x_1^j, x_2^j, \ldots, x_n^j$ is produced by P_n at the end of the $(j+n-1)$st time interval. One time interval later, that is, at the end of the $(j+n)$th time interval, P_n produces as output the maximum of $x_1^{j+1}, x_2^{j+1}, \ldots, x_n^{j+1}$.

3.4 Analysis

For definiteness, suppose that the function f^j is of the form given by Equation (2). It is clear that, for this function, the ratio of x_1^j to the maximum of x_1^j, x_2^j, …, x_n^j could be $O(1/n^w)$, in the worst case. Since the sequential computer returns x_1^j as the maximum, while the parallel computer obtains the exact maximum, using n processors instead of one yields an $O(n^w)$ improvement in the quality of the solution.

4 Conclusion

Parallelism was invented in order to speed up computations. Today, the principal purpose for using parallel computers remains the execution of computations that are too slow when performed sequentially. The overwhelming majority of theoretical and empirical analyses of parallel algorithms use the speedup provided by these algorithms as a measure of their goodness.

Another justification for using parallel computers, however, is the *quality* of the solution obtained by a parallel algorithm. It was shown in this paper that for the problem of computing (in a real-time environment) the maximum of a nonlinear feedback function over a given range, a parallel computer can deliver a solution that is better than any solution computed sequentially. In the worst case (from the point of view of sequential computation), the ratio of the solution obtained by the parallel algorithm to that obtained by the best possible sequential algorithm is superlinear in the number of processors used. It is especially interesting to note in this regard that this effect would certainly be magnified if the output of each computation were to be fed as input to the next computation (namely, if the maximum of $x_1^j, x_2^j, \ldots, x_n^j$ were to serve as the initial value x_0^{j+1}).

As pointed in [1], many computational problems are *inherently parallel*: If the *available* number of processors is smaller than the number of processors *required* to solve one of these problems (even if the difference is *one* processor), then the running time of the parallel algorithm is no better than that of the best sequential algorithm for the same problem [3]. Some problems, by contrast, are believed to be *inherently sequential*: No efficient parallel algorithm is known for solving any of these problems [28]. Real-time computation allows a different look at (apparently) inherently sequential problems. Suppose that a problem can be solved optimally in n (consecutive) time units. Further, let a new such problem be received by some computer system every time unit. The computer system is to process each new problem as soon as it arrives and produce its solution no later than n time units after receiving the problem. (These conditions are not unlike those established in Sections 3.1.2.) The parallel pipeline computer of Section 2.2.2 uses n processors to solve m such problems in $(m-1)+n$ time units. After an initial delay of n time units, an answer is produced every time unit. The parallel computer, therefore, meets the requirements of the problem. Furthermore, these computations (supposed to be inherently sequential) now seem to require constant time. On the other hand, it is clear (and paradoxical) that a sequential computer is hopelessly inadequate to solve these problems.

Other computational settings need to be explored for further measures to evaluate parallel algorithms. A candidate paradigm is one in which the data needed by an algorithm can be acquired from one of several sources. Each source holds a set of inputs sufficient by itself to solve the problem at hand. The inputs held by one particular source lead to a solution that is 'better' than any solution reached by using data from another source. At any given time, a single processor can acquire data from exactly one source. Furthermore, a source that is not selected for providing input to the algorithm ceases to exist (and its data can no longer be retrieved). In this paradigm, a sequential computer can find the best solution with probability $1/n$, where $n \geq 1$ is the number of sources. A parallel computer with n processors, on the other hand, assigns one processor to each source, and is therefore guaranteed to arrive at the best solution.

A variant to the paradigm described in the previous paragraph is one in which *all* sources need to be monitored simultaneously in order to obtain the best solution. Here, using a parallel computer with as many processors as there are sources (namely, n) is the only guarantee of success. This remains true even if—contrary to the basic assumption articulated at the beginning of Section 2.2—we allowed the sequential computer to use a processor that is n times faster than each of the processors on the parallel computer. When $n = 2$, a colorful illustration of the paradigm is the *pursuit and evasion on a ring* example presented in [1]. In this version, an entity A is in pursuit of another entity B on the circumference of a circle, such that A and B move at the same speed; clearly, A *never* catches B. Now, suppose that two entities C and D are in pursuit of entity B on the circumference of a circle. Each of C and D moves at $1/k$ the speed of A (and B), where k is a positive integer larger than 1. In this case, C and D *always* catch B. The present paradigm is another instance of inherently parallel problems in which it is the *parallelism* offered by the parallel computer that matters, rather than its speed [16]. Do other computational paradigms exist in which it is possible for parallel computers to obtain better solutions to computational problems than sequential ones?

Acknowledgments

I am grateful to Henk Meijer and Naya Nagy for helpful suggestions.

References

[1] S.G. Akl, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[2] S.G. Akl, Secure file transfer: A computational analog to the furniture moving paradigm, *Proceedings of the Conference on Parallel and Dis-*

tributed Computing and Systems, Cambridge, Massachusetts, November, 1999, 227–233.

[3] S.G. Akl, The design of efficient parallel algorithms, in: *Handbook on Parallel and Distributed Processing*, J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, Eds., Springer Verlag, Berlin, 2000, 13–91.

[4] S.G. Akl and S.D. Bruda, Parallel real-time optimization: Beyond speedup, *Parallel Processing Letters*, Vol. 9, No. 4, December 1999, 499–509.

[5] S.G. Akl and S.D. Bruda, Parallel real-time cryptography: Beyond speedup II, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 2000.

[6] S.G. Akl and S.D. Bruda, Parallel real-time numerical computation: Beyond speedup III, *International Journal of Computers and their Applications*, Special Issue on High Performance Computing Systems, Vol. 7, No. 1, March 2000, 31–38.

[7] S.G. Akl and L. Fava Lindon, Paradigms admitting superunitary behaviour in parallel computation, *Parallel Algorithms and Applications*, 11, 1997, 129–153.

[8] A. Bestavros and V. Fay-Wolfe, Eds., *Real-Time Database and Information Systems*, Kluwer Academic Publishers, Boston, 1997.

[9] L. Boxer and R. Miller, Dynamic computational geometry on meshes and hypercubes, *Journal of Supercomputing*, 3, 1989, 161–191.

[10] L. Boxer and R. Miller, Parallel dynamic computational geometry, *Journal of New generation Computer Systems*, 2(3), 1989, 227–246.

[11] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice Hall, Englewood Cliffs, New Jersey, 1998.

[12] S.D. Bruda and S.G. Akl, On the data-accumulating paradigm, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, Research Triangle Park, North Carolina, October 1998, 150–153.

[13] S.D. Bruda and S.G. Akl, The characterization of data-accumulating algorithms, *Proceedings of the International Parallel Processing Symposium*, San Juan, Puerto Rico, April 1999, 2–6.

[14] S.D. Bruda and S.G. Akl, A case study in real-time parallel computation: Correcting algorithms, *Proceedings of the Midwest Workshop on Parallel Processing*, Kent, Ohio, August 1999.

[15] S.D. Bruda and S.G. Akl, Towards a meaningful formal definition of real-time computations, *Proceedings of the Fifteenth International Conference on Computers and Their Applications*, New Orleans, Louisiana, March 2000, 274–279.

[16] S.D. Bruda and S.G. Akl, On the necessity of formal models for real-time parallel computations, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 2000.

[17] J. Casti, *Complexification: Explaining a Paradoxical World through the Science of Surprise*, HarperCollins, New York, 1994.

[18] P. Chaudhuri, Finding and updating depth first spanning trees of acyclic digraphs in parallel, *The Computer Journal*, 33, 1990, 247–251.

[19] P. Chaudhuri, *Parallel Algorithms: Design and Analysis*, Prentice Hall, Sydney, Australia, 1992.

[20] P. Chaudhuri, Parallel incremental algorithms for analyzing activity networks, *Parallel Algorithms and Applications*, 13(2), 1998, 153–165.

[21] F.Y. Chin and D. Houck, Algorithms for updating minimum spanning trees, *Journal of Computer and System Sciences*, 16, 1978, 333–344.

[22] J. Cohen and I. Stewart, *The Collapse of Chaos: Discovering Simplicity in a Complex World*, Viking, New York, 1994.

[23] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[24] P. Coveny and R. Highfield, *Frontiers of Complexity: The Search for Order in a Chaotic World*, Fawcett Columbine, New York, 1995.

[25] S. Even and Y. Shiloach, An on-line edge deletion problem, *Journal of the ACM*, 28, 1982, 1–4.

[26] G. Frederickson, Data structures for on-line updating of minimum spanning trees, *Proceedings of the ACM Symposium on Theory of Computing*, Boston, Massachusetts, April 1983, 252–257.

[27] J. Gleick, *Chaos: Making a New Science*, Penguin Books, New York, 1987.

[28] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo, *Limits to Parallel Computation*, Oxford University Press, New York, 1995.

[29] D. Harel, *Algorithmics: The Spirit of Computing*, Addison Wesley, Reading, Massachusetts, 1987.

[30] J.T. Havill and W.Mao, On-line algorithms for hybrid flow shop scheduling, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, Research Triangle Park, North Carolina, October 1998, 134–137.

[31] J. Horgan, *The End of Science*, Broadway Books, New York, 1996.

[32] T. Ibaraki and N. Katoh, On-line computation of transitive closure graphs, *Information Processing Letters*, 16, 1983, 95–97.

[33] S. Irani and A.R. Karlin, Online computation, in: D.S. Hochbaum, Ed., *Approximation Algorithms for NP-Hard Problems*, International Thomson Publishing, Boston, Massachusetts, 1997, 521–564.

[34] J. JáJá, *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, Massachusetts, 1992.

[35] H. Jung and K. Mehlhorn, Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees, *Information Processing Letters*, 27, 1988, 227–236.

[36] D.E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, Massachusetts, 1975.

[37] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, New York, 1976.

[38] H.W. Lawson, *Parallel Processing in Industrial Real-Time Applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[39] S. Levy, *Artificial Life: A Report from the Frontier Where Computers Meet Biology*, Vintage, New York, 1992.

[40] R. Lewin, *Complexity: Life at the Edge of Chaos*, Macmillan, New York, 1992.

[41] F. Luccio and L. Pagli, The p-shovelers problem (computing with time-varying data), *Proceedings of the Fourth Symposium on Parallel and Distributed Computing*, Arlington, Texas, December 1992, 188–193.

[42] F. Luccio and L. Pagli, Computing with time-varying data: Sequential complexity and parallel speed-up, *Theory of Computing Systems*, 31(1), 1998, 5–26.

[43] F. Luccio, L. Pagli, and G. Pucci, Three non conventional paradigms of parallel computation, *Lecture Notes in Computer Science*, 678, 1992, 166–175.

[44] P.B. Miltersen, S. Subramanian, J.S. Vitter, and R. Tamassia, Complexity models for incremental computation, *Theoretical computer Science*, 130, 1994, 203-236.

[45] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[46] S. Pawagi, A parallel algorithm for multiple updates of minimum spanning trees, *Proceedings of the International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Vol. III, 9–15.

[47] S. Pawagi and I.V. Ramakrishnan, An $O(\log n)$ algorithm for parallel update of minimum spanning trees, *Information Processing Letters*, 22, 1986, 223–229.

[48] G.J.E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms*, W.H. Freeman, New York, 1992.

[49] D.D. Sherlekar, S. Pawagi, and I.V. Ramakrishnan, $O(1)$ parallel time incremental graph algorithms, *Lecture Notes in Computer Science*, 206, 1985, 477–493.

[50] P.M. Spira and A. Pan, On finding and updating spanning trees and shortest paths, *SIAM Journal on Computing*, 4(3), 1975, 375–380.

[51] M. Thorin, *Real-Time Transaction Processing*, Macmillan, London, 1992.

[52] Y.H. Tsin, On handling vertex deletion in updating minimum spanning trees, *Information Processing Letters*, 27, 1988, 167–168.

[53] P. Varman and K. Doshi, A parallel vertex insertion algorithm for minimum spanning trees, *Lecture Notes in Computer Science*, 226, 1986, 424–433.

[54] M.M. Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon and Schuster, New York, 1992.